Laboratorium PTM

Programowanie w środowisku Microchip Studio – drugi program w języku C.

Celem ćwiczenia jest zaznajomienie ze środowiskiem i ilustracja najprostszych programów w języku C. W szczególności zaś przedstawienie wykorzystania pętli w programie na mikrokontroler ATmega328P

Zakład Systemów Informacyjno-Pomiarowych

IETiSIP, Wydział Elektryczny, PW

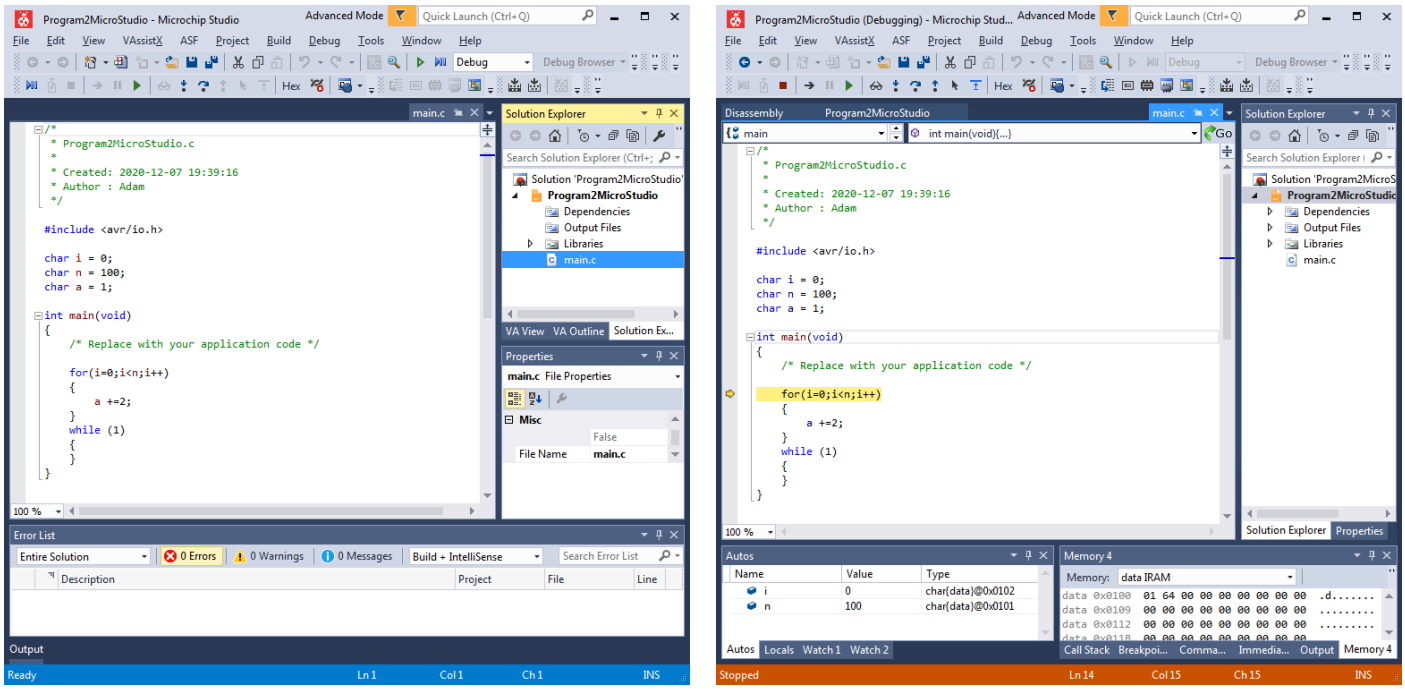
Program drugi w języku C:

Przy analizie kodów asemblerowych przydatna jest dokumentacja mikrokontrolera ATmega328P z listą rozkazów (i przypisanymi cyklami).

Drugi program w języku C dla mikrokontrolera ATmega328p niech ma postać taką jak na rysunku 1. Jak widać struktura programu również nie jest skomplikowana. Zawiera bowiem pojedynczą pętlę for z pojedynczą instrukcją dodawania. Wszystkie zmienne są typu **char** to znaczy, że zajmują w pamięci pojedynczy bajt i mogą przyjmować 256 wartości. Program rozpoczyna się włączeniem do programu nagłówka io.h:

```
#include <avr/io.h>
```

Nagłówek ten zawiera informacje o adresach rejestrów i wyprowadzeń używanego mikrokontrolera. Podczas tworzenia projektu wybrany został konkretny mikrokontroler a włączony nagłówek jest tu odpowiedzialny za przypisanie właściwych danych i dodatkowo zdefiniowanie nazw (literałów znakowych) dla rejestrów i wyprowadzeń konkretnego mikrokontrolera. Dzięki temu nie trzeba się szukać w dokumentacji specyficznych danych dla wybranego układu i co więcej można wykorzystywać ustandaryzowane nazwy dla poszczególnych elementów wybranego mikrokontrolera (np.: DDRB, PORTB, PINB).



Rysunek 1. Okno z programem do ćwiczenia.

Początkowe wartości zmiennych **i**, **n** i **a** zostały dobrane tak jak w typowej pętli **for**, przy założeniu 100 iteracji. Pętla **while** pozostała w kodzie, ale jej znaczenie jest wyłącznie techniczne, aby nie pozwolić na zakończenie programu i stracić możliwości podglądania wartości zmiennych. Kompilację i wygenerowanie plików binarnych można wykonać poprzez Build -> **Build Solution/Application** (F7), po ewentualnych zmianach kodu można skorzystać z opcji **Rebuild** (Ctrl+Alt+F7) a „zresetowanie" projektu można uzyskać poprzez **Clean Solution/Application**. W ćwiczeniu program można uruchamiać poprzez pasek narzędziowy i **Start Debugging** (F5) oraz **Stop Debugging** (Ctrl+Shift+F5). Identycznie można sterować wykonaniem programu wybierając odpowiednie opcje z menu **Debug**. W ćwiczeniach jednak wykorzystywać należy tryb krokowy:

Step Over (F10). Jest to tryb pracy krokowej bez wchodzenia do funkcji/procedur (wykonywane są one w jednym kroku). Jeżeli zaistnieje potrzeba przeanalizowania działania funkcji (szczególnie samodzielnie napisanej) wtedy można użyć trybu **Step Into** (F11). Możliwość wyjścia z procedury da w tym przypadku **Step out** (Shift+F11). Powyższe opcje są standardowe dla środowisk programistycznych. Po utworzeniu projektu i wpisaniu drugiego programu (F10) okno środowiska będzie wyglądać jak na rysunku 1 (z prawej strony). Wciskając kolejno F10 program wykonywał będzie kolejne instrukcje. Żółta strzałka wskazuje aktualną instrukcję do wykonania (wiąże się to z zawartością rejestru PC – Program Counter). W tym samym obszarze, w którym znajduje się strzałka można ustawiać pułapki (breakpoints). Wystarczy użyć lewego klawisza myszy. Pułapki są użyteczne, kiedy zaistnieje potrzeba dokładnej analizy pewnego fragmentu kodu (przy założeniu, że program jest złożony niż omawiany obecnie) z pominięciem pozostałej części.
Okno **Autos** (na dole po lewej stronie) zawiera zmienne zdefiniowane w programie wraz z informacją o położeniu (lokalizacji, adresie) zmiennej w pamięci danych. Architektura mikrokontrolerów ATmega jest architekturą Harwardzką z podziałem na pamięć danych i pamięć programu. W oknie **Memory** po prawej stronie można podglądać zawartość pamięci. Przy wyborze **IRAM** (Internal RAM) można podglądać aktualną zawartość pamięci danych. Znając adresy poszczególnych zmiennych w programie (okienko **Autos** i zmienne: i, n, a) można podglądać zmiany tychże zmiennych (w takt wykonywania programu - F10). Proszę zwrócić uwagę na kolejności umieszczania zmiennych w pamięci danych.

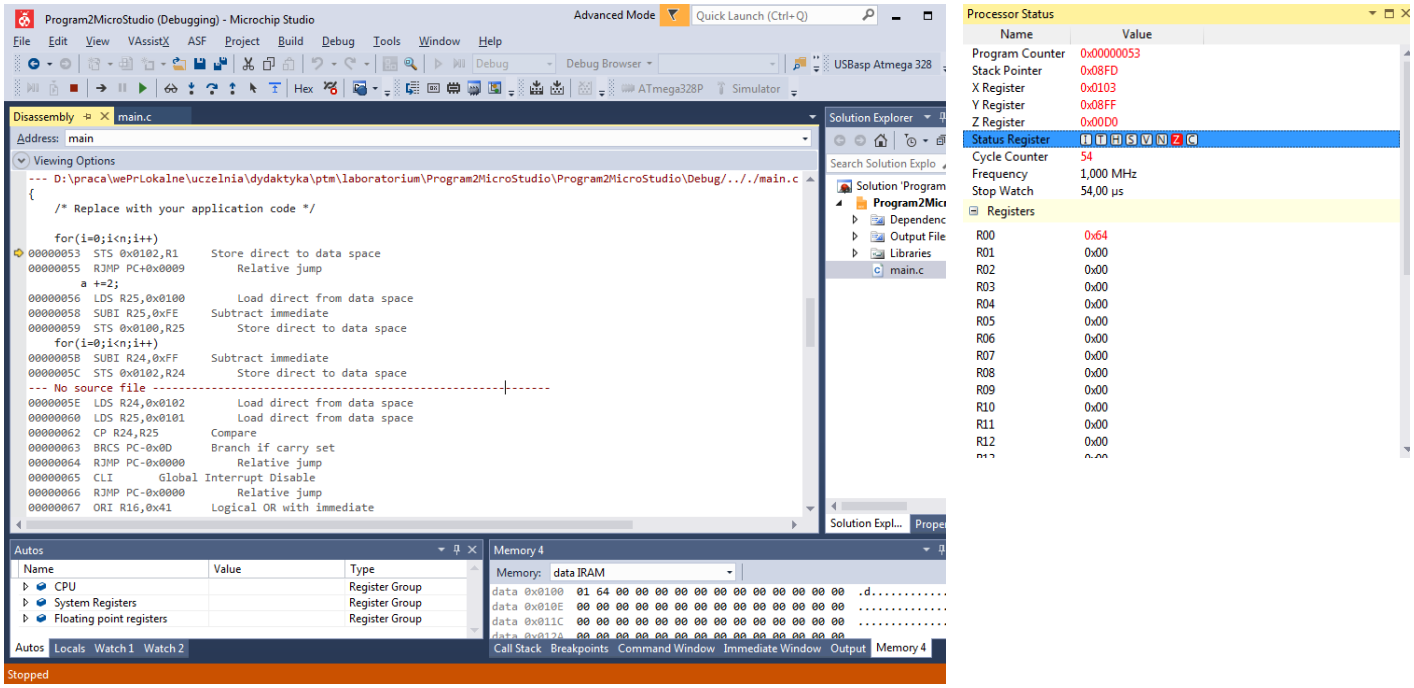
Figure 7-2. AVR CPU General Purpose Working Registers

	7	0	Addr.	
	R0		0x00	
	R1		0x01	
	R2		0x02	
	…			
	R13		0x0D	
General	R14		0x0E	
Purpose	R15		0x0F	
Working	R16		0x10	
Registers	R17		0x11	
	…			
	R26		0x1A	X-register Low Byte
	R27		0x1B	X-register High Byte
	R28		0x1C	Y-register Low Byte
	R29		0x1D	Y-register High Byte
	R30		0x1E	Z-register Low Byte
	R31		0x1F	Z-register High Byte

Rysunek 2. Rejestry ogólnego przeznaczenia mikrokontrolera ATmega328P.

UWAGA! Omawiany program został umieszczony w pamięci pod konkretnym adresem i wykorzystano w nim pewne konkretne rejestry ogólnego przeznaczenia Rxx (w programie są to rejestry R24 i R25 o adresach odpowiednio 0x18 i 0x19 – przedrostek 0x oznacza zapis heksadecymalny czyli szesnastkowy – rysunek 2). Zarówno adres początkowy programu w pamięci, jak i wykorzystane rejestry mogą się różnić w różnych kompilacjach. Dlatego podczas analizy własnych programów należy uwzględnić zarówno możliwość innego adresu początkowego zasadniczego kodu programu, inne zestawy rejestrów oraz różne położenie zmiennych w pamięci danych. Wiąże się to z koniecznością obserwacji innych obszarów pamięci.

Szczegóły wykonania programu można poznać wybierając (przy uruchomionym programie) **Debug**->**Windows**-> **Disassembly** i otrzymując w wyniku asemblerowy kod programu (rysunek 3).



Rysunek 3. Okno z kodem asemblerowym programu.

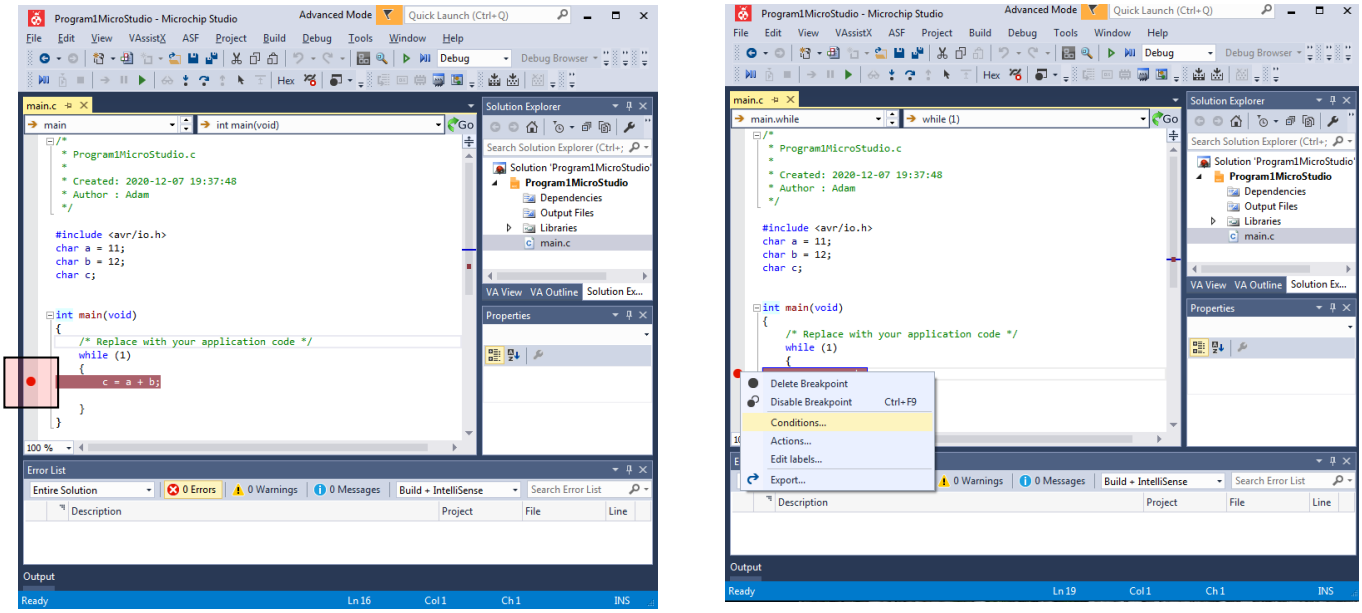
W języku asemblera również istnieje możliwość wykonywania programu w trybie krokowym. Jednakże w tym przypadku wykonywanie programu odbywa się instrukcja po instrukcji asemblerowej (NIE zaś instrukcja po instrukcji w języku C). Wartość prezentowana, jako pierwsza z lewej strony do adres w pamięci programu, czyli zawartość rejestru PC – Program Counter. Wygenerowany kod asemblerowy pochodzi z szablonu/schematu środowiska programistycznego. Fragment, który będzie w obszarze zainteresowania znajduje się poniżej komentarza z informacją o pliku źródłowym, w którym się znajduje tj. ---… /main.c. Komentarze dodane przez środowisko wyjaśniają dokładnie znaczenie poszczególnych instrukcji asemblerowych.

Wygenerowany kod asemblerowy drugiego programu:
Na początek może wydawać się niejasna przyczyna, dla której dwukrotnie w kodzie pojawia się instrukcja pętli **for**. Jest tak dlatego, że w kodzie pętli są trzy operacje. Po pierwsze inicjacja zmiennej indeksującej **i**, po drugie warunek końcowy i wreszcie aktualizacja wartości indeksu. Charakterystyczne jest to, że o ile warunek pętli i aktualizacja są wykonywane każdorazowo raz na iterację pętli **for**, to inicjacja wykonywana jest jedynie raz na początku. Stąd ta specyficzna organizacja kodu asemblerowego. Zasadnicza część programu umieszczona jest w pamięci (programu) począwszy od adresu 0x00000053. Co charakterystyczne adresy pamięci programu wcale nie są rozłożone monotonicznie. **Wynika to z faktu, że niektóre instrukcje zajmują w pamięci programu bajt a inne dwa bajty??????**

```
--- .../main.c
{
	/* Replace with your application code */
;Zmiennym i, a oraz n przypisane są adresy w pamięci danych odpowiednio: 0x102, 0x100 i 0x101.
;W programie wykorzystywane są rejestry ogólnego przeznaczenia mikrokontrolera, tu: R24
;i R25 oraz na wstępie R1.
	for(i=0;i<n;i++)
;Poniższa instrukcja zeruje zmienną i. W tym celu wykorzystany został rejestr R1, który uprzednio został wyzerowany. Jeśli by prześledzić cały kod asemblerowy, na jego początku napotkamy instrukcję „zerującą” CLR R1.
00000053  STS 0x0102,R1        ;Store direct to data space
;Na razie pomijamy a+=2 gdyż należy uprzednio sprawdzić warunek pętli for tj. i<n. W tym celu realizowany jest skok pod adres 0x0055+0x0009=0x005E
```

```
00000055  RJMP PC+0x0009          ;Relative jump
            a +=2;
;Zmienna a ładuje się do rejestru R25
00000056  LDS R25,0x0100          ;Load direct from data space
;Poniższa instrukcja efektywnie odpowiada za operację a = a + 2, pomimo iż faktycznie
;realizuje odejmowanie. Dzieje się tak za sprawą własności formatu zapisu danych U2 i
możliwości realizacji operacji a = a – (-2). 0xFE (szesnastkowo) = -2 (dziesiętnie).
;Powstaje pytanie a dlaczego tak ? Otóż odpowiedź nie jest skomplikowana. Jeśli przejrzymy
listę rozkazów mikrokontrolera ATmega328P okaże się, że ma on dwie instrukcje dodawania
wielkości bajtowych ADD i ADDC. Obydwie pobierają argumenty wejściowe poprzez rejestry,
które trzeba uprzednio załadować. W efekcie ADD lub ADDC zajmuje 1 cykl zegarowy a
instrukcja LDS zajmuje 2 cykle. Instrukcję tę należy wykonać dwukrotnie dla obu rejestrów
co w efekcie daję łącznie 5 cykli. Zamiast tego wykonywana jest operacja odejmowania stałej
od wartości rejestru co zajmuje 1 cykl plus pojedyncze ładowanie rejestru (LDS) zajmujące 2
cykle czyli łącznie 3 cykle (taka optymalizacja czasu wykonania :)). Na liście
mikrokontrolera ATmega328P jest jeszcze instrukcja ADIW dla wielkości dwubajtowych.
Jednakże wykonuje ona operacje kolejno dla dwóch bajtów, co zajmuje 4 cykle zegarowe.
00000058  SUBI R25,0xFE          ;Subtract immediate
;Zwiększona wartość zmiennej a ląduje z powrotem pod adres tej zmiennej.
00000059  STS 0x0100,R25          ;Store direct to data space
      for(i=0;i<n;i++)
;W rejestrze R24 umieszczona została uprzednio zmienna i. Analogicznie do powyższej
instrukcji SUBI w tym miejscu wykonywana jest operacja i = i + 1 a dokładnie i = i -(-1)
gdyż 0xFF (szesnastkowo) = -1 (dziesiętnie).
0000005B  SUBI R24,0xFF           ;Subtract immediate
;Zwiększona wartość zmiennej i ląduje z powrotem pod adres tej zmiennej.
0000005C  STS 0x0102,R24          ;Store direct to data space
--- No source file ------------------------------------------------------------
;Do rejestru R24 ładuje się zmienna i.
0000005E  LDS R24,0x0102          ;Load direct from data space
;Do rejestru R25 ładuje się zmienna n.
00000060  LDS R25,0x0101          ;Load direct from data space
;Porównanie zmiennych i oraz n. W praktyce wykonuje się operacja R24-R25 czemu towarzyszy
warunkowe ustawienie flag (zestaw bitów, z którego tym razem ważny jest bit/flaga
przeniesienia). Bit przeniesienia jest ustawiony każdorazowo, kiedy R24<R25 lub jak w
programie i<n.
00000062  CP R24,R25              ;Compare
;Instrukcja skoku warunkowego. Jeśli ustawiona jest flaga przeniesienia sterowanie
programem zostanie przekierowane pod adres PC-0x0D czyli 0x0063 – 0x0D = 0x0056. Jeśli
flaga nie zostanie ustawiona sterowanie przechodzi do następnej linii kodu gdzie znajduje
się implementacja nieskończonej pętli while(1).
00000063  BRCS PC-0x0D            ;Branch if carry set
;Nieskończona pętla while(1)
00000064  RJMP PC-0x0000          ;Relative jump
;Poniższe instrukcje wynikają z szablonu kodu dla mikrokontrolera
00000065  CLI                     ;Global Interrupt Disable
00000066  RJMP PC-0x0000          ;Relative jump
00000067  ORI R16,0x41            ;Logical OR with immediate
```

Co się stanie jeśli n zmienione zostanie na 150 ? Można zmienić definicję zmiennych na **unsigned char**, następnie na **signed char** i porównać eksperymenty. Czy program wykona się prawidłowo? Obserwacje warto przeprowadzić z zastosowaniem pułapek, ang. breakpoints. Taką pułapkę można „zastawić" klikając z lewej strony w szarym obszarze okna z kodem (rysunek 4). Dodatkowo wywołując menu kontekstowe do pułapki można dodatkowo ustawić warunek (**Conditions...**). Przydatne jeśli np.: pętla ma wykonać 1000 iteracji, a sprawdzeniu/analizie podlega ostatnie 5. Przy czym w tym przypadku istotna jest znajomość zakresu danych jakie „pomieści" zmienna typu **char**, **signed char**, **unsigned char** czy wreszcie **int**. Jaki jest domyślny typ zmiennych **signed char** czy **unsigned char** ?



Rysunek 4. Ustawianie pułapek.

Zadania do wykonania:
Proszę zmienić definicję zmiennych na **int** a zmienną **n** ustawić na 300. Pułapkę (breakpoint) można ustawić tak aby aktywowała się po 296 iteracjach. Jaki jest domyślny typ **int**, **signed** czy **unsigned** ?
Proszę policzyć sumę albo maksimum albo minimum albo średnią (całkowitoliczbową) z tablicy. W tym celu należy zdefiniować kilkuelementową tablicę zmiennych w pierwszym przypadku **char,** a w drugim **int** i zainicjować wartościami.
Ręcznie np:

```
tablica[0] = 1000;
tablica[1] = 2000;
```

…
Albo automatycznie

```
for(i=0;i<N;i++)
      tablica[i] = wartosc;
```

Następnie zdefiniować zmienną do przechowania odpowiedniej wartości i w pętli wykonać stosowne czynności tak aby wyznaczyć konkretną wartość. Proszę przeanalizować wynikowy kod asemblerowy. Na co należy zwrócić uwagę wyliczając ww. wartości ?

Dodatek A:

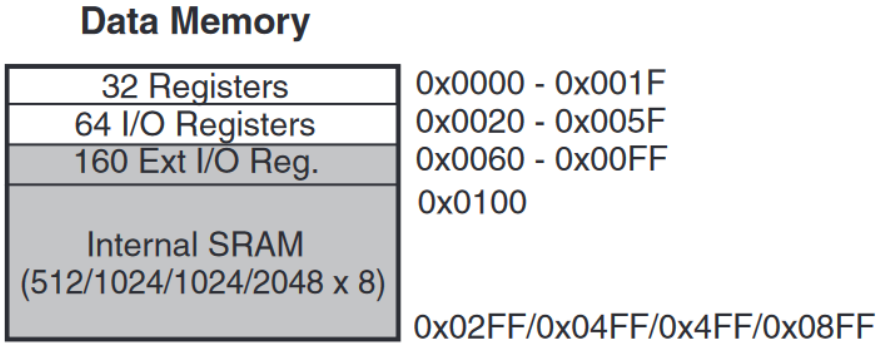
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x15 (0x35)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0	
0x14 (0x34)	Reserved	–	–	–	–	–	–	–	–	
0x13 (0x33)	Reserved	–	–	–	–	–	–	–	–	
0x12 (0x32)	Reserved	–	–	–	–	–	–	–	–	
0x11 (0x31)	Reserved	–	–	–	–	–	–	–	–	
0x10 (0x30)	Reserved	–	–	–	–	–	–	–	–	
0x0F (0x2F)	Reserved	–	–	–	–	–	–	–	–	
0x0E (0x2E)	Reserved	–	–	–	–	–	–	–	–	
0x0D (0x2D)	Reserved	–	–	–	–	–	–	–	–	
0x0C (0x2C)	Reserved	–	–	–	–	–	–	–	–	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	101
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	101
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	101
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	100
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	100
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	101
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	100
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	100
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	100
0x02 (0x22)	Reserved	–	–	–	–	–	–	–	–	
0x01 (0x21)	Reserved	–	–	–	–	–	–	–	–	
0x0 (0x20)	Reserved	–	–	–	–	–	–	–	–	

Note:
1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The ATmega48A/PA/88A/PA/168A/PA/328/P is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for ATmega88A/88PA/168A/168PA/328/328P.
6. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA/328P

Rysunek A.1: Rejestry specjalne (I/O Registers) mikrokontrolera ATmega328P wraz z ich adresami.

Proszę zwrócić uwagę (rysunek A.1), że przy dostępie bezpośrednim (np.: instrukcje IN, OUT) podaje się adresy względem początku bloku rejestrów specjalnych (I/O Registers). Stąd przy dostępie do rejestru PORTB należy użyć adresu 0x04. Jeżeli natomiast do tego samego rejestru chcemy się odwołać za pomocą instrukcji LD czy SD należy użyć adresowania jednolitego wynikającego z organizacji pamięci danych (Rysunek A.2). Stąd w tym przypadku adres dla ww. rejestru PORTB wynosi 0x04+0x20 = 0x24.

Figure 8-3. Data Memory Map



Rysunek A.2: Organizacja adresowa pamięci danych mikrokontrolera ATmega328P.

****

Adresy rejestrów dla mikrokontrolera ATmega328P:
DDRB: 0x04 (0x24)
PORTB: 0x05 (0x25)
PINB: 0x03 (0x03) Port B Input
R24: 0x18
R25: 0x19
IRAM – Internal RAM
RS jest na porcie D
I2C jest na porcie C
SPI jest na porcie B